https://doi.org/10.31891/2219-9365-2024-78-24
УДК: 681.3.06

МИКИТИН Андрій
Національний Університет "Львівська Політехніка"
https://orcid.org/0009-0002-6645-3224
e-mail: andrik.mykytyn@gmail.com

РОЗБІР ПРИНЦИПІВ РОБОТИ РЕАКТИВНОГО ПРОГРАМУВАННЯ НА ОСНОВІ МОВИ JAVA ТА SPRING BOOT ФРЕЙМВОРКУ У ПОРІВНЯННІ ІЗ ОСНОВНИМИ ПІДХОДАМИ ДО РОЗРОБКИ ДОДАТКІВ

У статті описані та проаналізовані загальноприйняті способи розробки BE сервісів та їхнє порівняння до реактивного програмування на основі мови програмування Java та Spring Framework. Наведені приклади імплементації усіх трьох підходів. Також описані принципи роботи основних конструкцій реактивного програмування на мові Java.
Ключові слова: реактивне програмування, додаток, потік.

MYKYTYN Andrii
Lviv Polytechnic National University

ANALYSIS OF THE PRINCIPLES OF REACTIVE PROGRAMMING BASED ON THE JAVA LANGUAGE AND THE SPRING BOOT FRAMEWORK IN COMPARISON WITH THE MAIN APPROACHES TO APPLICATION DEVELOPMENT

This topic is relevant because modern systems need to handle large volumes of data, load balancing, easy expansion and small amounts of resources on the cloud. Load balancing requires an increase in the resources used at the hardware level, but this is only a temporary solution unless changes are made to improve it at the code level. The need today is not just to get an even result, but to get it quickly with the least amount of resources used. The biggest problem with the performance of BE applications is the inefficient use of threads at the server level. During the execution of one thread, there are moments when it is idle waiting for a response from various auxiliary resources or the database. This time can be effectively used for lossless information processing to perform the main functionality of the stream.

Reactive programming is built around two structures, Mono<T>[1] and Flux<T>. When Mono<T> is executed, all depicted events occur in one thread, the operator processes the object and returns it to the Subscriber object. Returns an error on error. The principle of operation of Flux<T> is identical to the operation of Mono<T>, the only difference is the number of objects that are processed. Both constructs implement methods Publisher and CorePublisher, which inherits from Publisher. These abstractions contain only one method subscribe(CoreSubscriber<? super T> subscriber). This method is called only when calling a method that returns a Mono<T> or Flux<T> object. From which we can conclude that using a reactive approach to writing logic, we subscribe to the result of the execution of the method. In this way, it is possible to track when a method is waiting for a result and requires a thread to continue executing logic. This approach is implemented according to the Observer pattern principle. Reactive programming also uses a pattern like the Interator to get a thread to execute the next method or functionality that doesn't necessarily belong to the user thread. In this way, the framework allows not only to provide a thread to execute when the wait is over and we have a result, but also to free up that thread to execute other methods that do not require waiting.

Reactive programming is a design approach that uses asynchronous programming logic to execute code. Reactive applications that are non-blocking by design and allow the application to scale with a small fixed number of threads and lower memory requirements, while making the best use of available computing power. This sets it apart from the user thread blocking approach and the asynchronous approach to data processing, which either block the thread without effectively using it.
Keywords: reactive programming, application, BE, thread, subscribe,

Постановка проблеми у загальному вигляді та її зв'язок із важливими науковими чи практичними завданнями

При розробці комплексних та ресурсовитратних рішень для малого бізнесу чи start-up-пів часто виникає потреба використання існуючих ресурсів із найбільшою ефективністю без збільшення апаратних можливостей сервера. Наприклад, відправка емейлів, збереження файлів, редагування відео, тощо. Описаний підхід повністю вирішує цю проблему та надає можливості обробки більших об'ємів інформації без збільшення апаратних можливостей.

Окремо можемо виділити вимоги, що потребують простоти в імплементації та описання документації. Для спрощення розуміння проблеми у статті буде розглянуто буде розглянуто підхід, який частково вирішує проблему шляхом асинхронного виконання коду, що є частково ефективно, але із використанням складної імплементації.

Аналіз досліджень та публікацій

Використання реактивного програмування в системах із великим навантаженням здебільшого асоціюється із певними перевагами порівняно із блочними реквестами та асинхронним програмуванням[2-3]. Реактивні системи мають важливе практичне застосування через свою спроможність негайно реагувати

на події та зміни у реальному часі. Це забезпечує системи, що важливо для користувачів та для сценаріїв, де кожна затримка неприйнятна.

Однією з ключових переваг реактивного програмування є можливість обробки подій асинхронно, що дозволяє уникати блокування потоку виконання. У порівнянні із блочними запитами, де операції можуть чекати завершення одна одної, реактивні системи можуть обробляти багато операцій паралельно. Це робить їх ефективнішими в умовах високого навантаження та дозволяє використовувати ресурси більш оптимально[3].

Крім того, реактивне програмування сприяє створенню систем, які можуть легко масштабуватися. Здатність реактивних систем адаптуватися до збільшення чи зменшення навантаження дозволяє забезпечити стабільну та ефективну роботу системи в умовах змінюваних обсягів даних чи користувацького трафіку.

Формулювання цілей статті

Метою роботи є: Дослідити принципи роботи реактивного програмування у порівнянні із загальноприйнятими підходами.

Виклад основного матеріалу

Ця тема є актуальною, оскільки сучасні системи повинні обробляти великі об'єми даних, балансування навантаження, легкого розширення та малого об'єму ресурсів на клауді. Балансування навантаження вимагає збільшення використовуваних ресурсів на апаратному рівні, але це є тільки тимчасове рішення, якщо не вносити зміни для покращення на рівні коду.Потреба сьогодення це не просто отримати рівний результат, а отримати його швидко із найменшою кількістю використаних ресурсів.

Найбільшою проблемою ефективності роботи ВЕ додатків є неефективне використання потоків на рівні сервера. Під час виконання одного потоку є моменти коли він простоює в очікуванні відповіді він різних допоміжних ресурсів або бази даних. Цей час можна ефективно використовувати для обробки інформації без втрат для виконання основного функціонала потоку. Розглянемо приклад коду із використанням блокування потоків на мові Java з використанням Spring Framework:

```
@GetMapping("user/{id}")
public User getUserInfo(@PathVariable final Long id){
   final User user = userService.getUserById(id);
   final UserMetaData userMetaData = userMetaDataService.getUserMetaData(id);
   user.setMetaData(userMetaData);
   return user;
}
```

Якщо проаналізувати виконання даного коду та зобразити це у вигляді Request diagram (Рис. 1) то можна побачити, що місця де потік блокується та очікує на відповідь від бази даних, що не є критично, якщо система не є навантаженню.

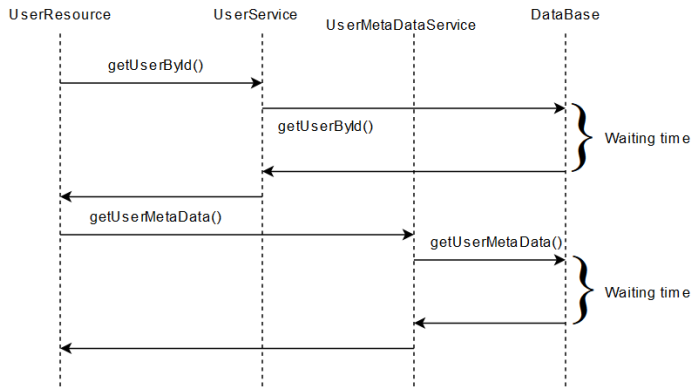

Рис. 1. Request діаграма роботи сервісу із блокування потоків

Це час можна використати для виконання інших функцій, або для розвантаження важких операцій які вимагають багато ресурсів сервера. Проблему із неефективним використанням потоків можна вирішити за допомогою асинхронного виконання операцій і об'єднання їхнього результату. Розглянемо приклад коду на мові Java із використанням Spring Framework:

```
@GetMapping("user/{id}")
public User getUserInfo(@PathVariable final Long id){
```

```
final CompletableFuture<User> userAsync = CompletableFuture.supplyAsync(() ->
userService.getUserInfo(id));
final CompletableFuture<Preferences> metaDataAsync = CompletableFuture.supplyAsync(() ->
userMetaDataService.getUserMetaData(id));
final CompletableFuture<Void> userMetaDataAsync = CompletableFuture.allOf(userAsync,
metaDataAsync);
userMetaDataAsync .join();
//return user object
final User user = userAsync.join();
final UserMeTaData userMetaData = metaDataAsync .join();
user.setMetaData(userMetaData );
return user;}
```

У даному прикладі виклики виконуються паралельно, але зрештою об'єднання результату getUser() у getUserMetaData() блокується, доки не буде отримано відповіді від обох викликів. Відповідності ефективне використання двох дочірніх потоків для виконання операцій, але досі присутнє блокування основного потоку. В додачу до цього складність в розумінні коду та відсутня простота у читанні такого коду[4].

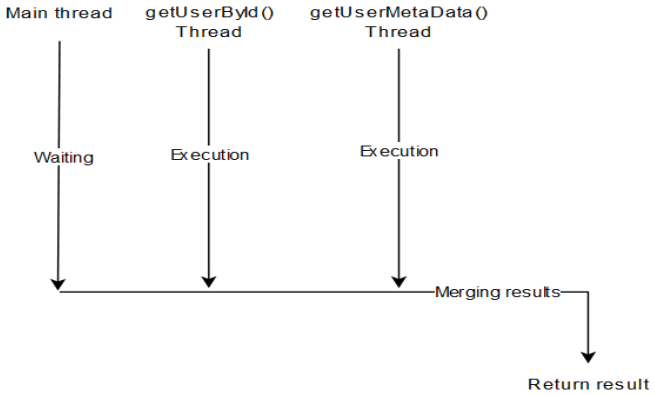

Рис. 2. Схематичне зображення асинхронної роботи сервісу

Розглянемо приклад імплементації та роботи коду із використанням реактивного програмування у мові програмування Java та Spring Framework:

```
@GetMapping("user/{id}")
public Mono<User> getUserInfo(@PathVariable final Long id ){
  return userService.getUser(id)
  .zipWith(userMetaDataService.getUserMetaData(id))
  .map(row -> {
    final User user = row.getT1();
    user.setUserMetaData(row.getT2());
    return user;
  });
}
```

Якщо зобразити виконання цього коду на діаграмі то ми побачимо таку картину:

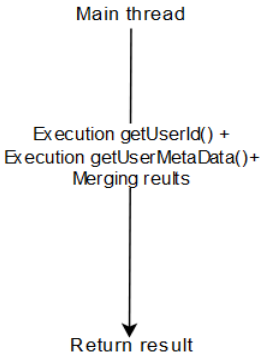

Рис. 3. Схематичне зображення роботи реактивного додатка

На рис. 3 всі операції виконуються в одному потоці, без простоювання. Це досягнуто за допомогою уникнення блокування потоку на моменті очікування відповіді на запит до бази даних, як зображено на рис. 1 і використовувати цей час для початку виконання getUserMetaData() методу, який встигне пройти до власного запиту до бази даних і поки він буде на нього очікувати, перший метод завершить своє виконання отримавши відповідь та поверне потік для другого методу, який своєю чергою вже буде мати відповідь від бази даних.

Взявши до уваги вище згадані приклади та порівняти ефективність виконання потоків, то можна зробити висновок, що реактивний підхід є найбільш ефективним з усіх загадних. Варто також пам'ятати, про те, що цей підхід є найбільш ефективним при обробці великих об'ємів даних, якщо система не передбачає цього, то не варто зловживати цим, оскільки це може вплинути на поточну продуктивність додатка в гіршу сторону, оскільки час на обробку потоків та їхнє перекидання між завданням займає час та ресурси.

Якщо ви помітили, останній приклад коду використовує конструкцію Mono<T> (рис. 4), що є однією із двох основних конструкцій результату виконання реактивного методу. Ця конструкція використовується для обробки одиничного результату виконання методу(0 - 1) , тобто метод повинен повертати один об'єкт за одне виконання. Також використовується конструкція Flux<T>(рис. 5) для обробки результату, який представлений колекцією, тобто один і більше об'єкт за одне виконання методу(0 - N). Розглянемо принципи роботи цих двох конструкцій:

Mono

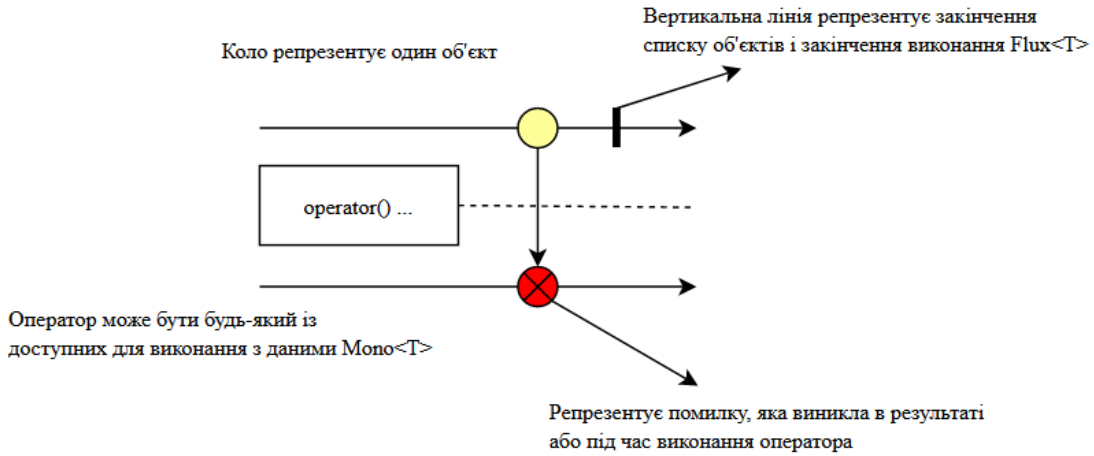

Рис.4. Схематичне зображення роботи Mono<T> конструкції

При виконанні Mono<T> всі зображені події відбуваються в одному потоці, оператор обробляє об'єкт та повертає його до об'єкту Subscriber. У випадку помилки повертає помилку.

Flux

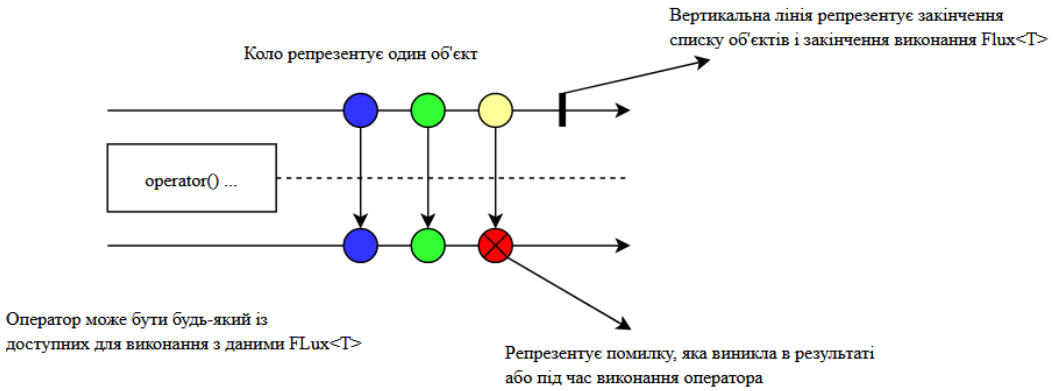

Рис. 5. Схематичне зображення роботи Flux<T> конструкції

Принцип роботи Flux<T> ідентичний до роботи Mono<T> єдина відмінність це кількість об'єктів, які обробляються. Обидві конструкції імплементують методи Publisher та CorePublisher, який наслідується від Publisher. Ці абстракції містять тільки один метод subscribe(CoreSubscriber<? super T> subscriber). Цей метод викликається тільки при виклику методу який повертає Mono<T> або Flux<T> об'єкт. Звідки можна зробити висновок, що використовуючи реактивний підхід до написання логіки ми підписуємося на

результат виконання методу. У такий спосіб можливо відстежувати, коли метод своє очікування на результат і вимагає потоку для продовження виконання логіки. Цей підхід імплементований за принципом Observer патерну. У реактивному програмуванні також використовується такий патерн, як Interator для отримання потоку для виконання наступного по черзі методу, або функціоналу, який не обов'язково належить до користувацького потоку. У такий спосіб фреймворк дозволяє не тільки надавати потік для виконання коли очікування закінчилося та у нас є результат, а також звільняти цей потік для виконання інших методів, які не вимагають очікування.

Висновки з даного дослідження і перспективи подальших розвідок у даному напрямі

Реактивне програмування — це підхід до проєктування, який використовує логіку асинхронного програмування для виконання коду. Реактивні додатки, які за своєю структурою не блокують виконання потоку і надають можливість масштабувати додаток із невеликою фіксованою кількістю потоків і меншими вимогами до пам’яті, водночас найкращим чином використовуючи доступну обчислювальну потужність. Це виділяє його серед підходу із блокування користувацького потоку та асинхронного підходу до обробки даних, які в будь-якому випадку блокують потік без ефективного його використання. Але також не варто зловживати цим підходом через його складність роботи на рівні фреймворку.

Література

1. Oleh D. Spring Reactive [Електронний ресурс] / D. Oleh, Л. Ігор. – 2024. – Режим доступу до ресурсу: https://spring.io/reactive.
2. Koushik K. Reactive programming with Java - full course [Електронний ресурс] / Kothaga Koushik // Java Brains. – 2022. – Режим доступу до ресурсу: https://www.youtube.com/watch?v=EExlnnq5Grs&ab_channel=JavaBrains.
3. Shalu M. Reactive Programming (Java) [Електронний ресурс] / Malhotra Shalu // We are Community. – 2023. – Режим доступу до ресурсу: https://wearecommunity.io/communities/india-java-user-group/articles/2623.
4. Anshul B. Asynchronous Programming in Java [Електронний ресурс] / B. Anshul, M. Eric // The Baeldung. – 2024. – Режим доступу до ресурсу: https://www.baeldung.com/java-asynchronous-programming.

References

1. Oleh D. Spring Reactive [Elektronnyi resurs] / D. Oleh, L. Ihor. – 2024. – Rezhym dostupu do resursu: https://spring.io/reactive.
2. Koushik K. Reactive programming with Java - full course [Elektronnyi resurs] / Kothaga Koushik // Java Brains. – 2022. – Rezhym dostupu do resursu: https://www.youtube.com/watch?v=EExlnnq5Grs&ab_channel=JavaBrains.
3. Shalu M. Reactive Programming (Java) [Elektronnyi resurs] / Malhotra Shalu // We are Community. – 2023. – Rezhym dostupu do resursu: https://wearecommunity.io/communities/india-java-user-group/articles/2623.
4. Anshul B. Asynchronous Programming in Java [Elektronnyi resurs] / B. Anshul, M. Eric // The Baeldung. – 2024. – Rezhym dostupu do resursu: https://www.baeldung.com/java-asynchronous-programming.